

HYPERCUBE TECHNOLOGY

Jay W. Parker, Tom Cwik, Robert D. Ferraro, Paulett C. Liewer, Jean E. Patterson

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California 91109

Abstract: The JPL-designed MARKIII hypercube supercomputer has been in application service since June 1988 and has had successful application to a broad problem set including electromagnetic scattering, discrete event simulation, plasma transport, matrix algorithms, neural network simulation, image processing and graphics. Currently, problems that are not homogeneous are being attempted, and, through this involvement with "real world" applications, the software is evolving to handle the heterogeneous class problems efficiently.

Hypercube Computers and the Mark IIIfp System: The appearance of parallel computers comes through a natural coupling of two important factors. The first one is the clear *opportunity* presented by current VLSI technology, which determines the availability of powerful single-chip microprocessors and inexpensive memory. The second one is the pressing *need* for increased computational power in a wide variety of scientific and engineering fields. A particularly simple parallel computer consists of loosely coupled processors connected in a binary hypercube topology. A D -dimensional hypercube contains a total of $D2^{D-1}$ communication links, and 2^D nodes where each node is connected to D other nodes. One of the most useful characteristics of this topology is that the diameter of the network (the maximum number of links that a message has to travel between any source and any destination along the shortest path) is also equal to D . The JPL/Caltech Hypercube is now in its third generation of development. The current implementation in the series has been designated the Mark IIIfp Hypercube, and is based on Motorola and Weitek chip sets, as described below. A description of the earlier Mark III and several related parallel computers, their software environment, and sample applications approaches and codes may be found in [1]. Further details regarding parallel operating systems and applications may be found in [2].

Fig. 1 shows the general system description of a 32-node (5-dimensional) Mark IIIfp Hypercube system. The main components of the system are shown: a 32-node Mark IIIfp, the control processor, the Concurrent I/O (CIO) network and the disk subsystem. Each node in the Mark IIIfp is a powerful single-board computer which contains two independent processing elements, namely the *data processor* and the *I/O processor*. Each node contains its own local memory and a set of peripherals. In addition, each node contains 4 Megabytes of dynamic RAM accessible to both processors. Each node has a total of eight communication channels, one of which is reserved for communications outside the hypercube. Thus, the current architecture is limited to 128 nodes, or a 7-dimensional Mark IIIfp.

The data processor consists of a Motorola MC68020 CPU with a MC68882 floating point math coprocessor, two serial ports, and a printer port. The serial ports can be used to connect a terminal to the node to monitor the activity of the data processor. Associated with the data processor are 128K bytes of private no-wait-state static RAM. Programs run approximately 15% faster when stored within this memory space. The I/O processor consists of a Motorola MC68020 CPU, one serial port, and hardware to support the node-to-node communications within the hypercube. The serial port can also be used to connect a terminal to the node and monitor the activity of the I/O processor, independent of the data processor. Associated with the I/O processor are 64K bytes of private no-wait-state static RAM.

The floating point daughterboard incorporates a Weitek chip set and a serial port (the letters "fp" on the designation of the Mark IIIfp hypercube denote the active inclusion of the Weitek daughterboard into the architecture). The Weitek chip set consists of a sequencer, an integer processor, and a pipelined floating point processor, and is capable of performing from 1 to 10 MFLOPS. Typical applications written in C or FORTRAN with calls to CrOS communications achieve the lower end of this range. Many such applications maintain near-linear performance when scaled to larger hypercubes; one can expect real application performance for a 128 node hypercube to be on the order of 100-300 MFLOPS.

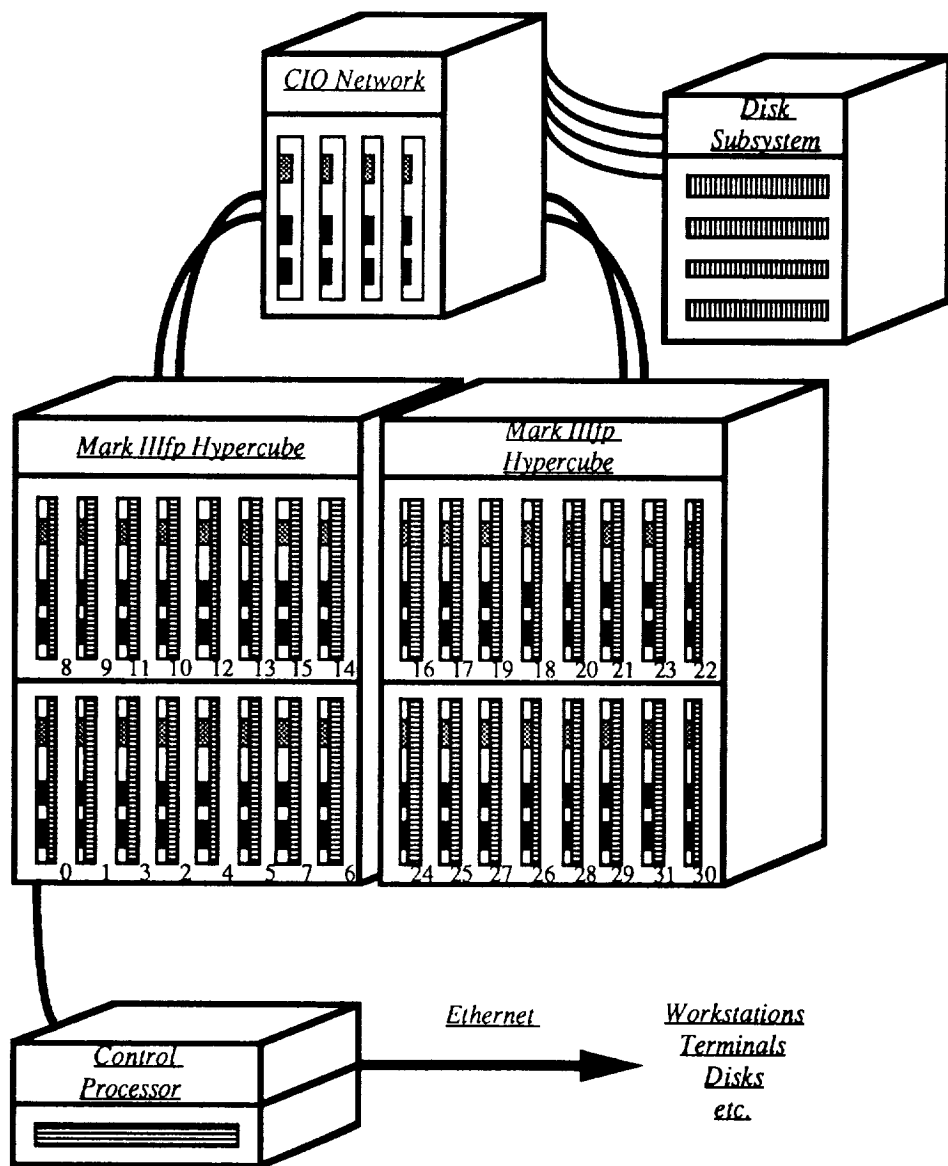


Figure 1. A 32-node (5-dimensional) JPL/Caltech Mark IIIfp Hypercube system. This figure shows the main components of the system: the hypercube, the control processor, the CIO network and the disk subsystem. The connections between these components are also shown, including the interface to the outside world through Ethernet.

Recently, the Weitek board has been upgraded to support 64-bit double-precision arithmetic. The serial port on the Weitek daughterboard can be used to connect a terminal to monitor the activity of the Weitek processor, independent of both the data processor and the I/O processor. The Weitek processor has access to its own private memory, which consists of 64K bytes of no-wait-state static RAM as well as a 128K byte code cache and a 64K byte data cache.

The Mark IIIfp is accessed from the external world through a host or control processor (CP) computer, which in turn communicates with the hypercube through a JPL-designed, special-purpose interface connected to the corner node (node 0) of the ensemble. The control processor is a Unix workstation based on the Motorola MC68020 microprocessor and currently is a Counterpoint System 19. The CP can be attached to peripheral devices such as disk drives, terminals, and printers and acts as an access controller mechanism to these devices for the entire hypercube. Since the control processor is based on the same

MC68020 microprocessor used in the node boards, the native compilers and linkers, as well as the full complement of Unix tools can be used for program development. These tools allow the construction of executable code on the control processor which ultimately runs on the nodes of the Mark IIIfp.

A concurrent I/O (CIO) node board resembles a Mark IIIfp's main board but has a single MC68020 processor and only four communication channels. In addition to the ability to form a hypercube network consisting of CIO nodes, a CIO board has a VME bus interface which allows the Mark IIIfp to communicate with disks, graphics devices or specialized processing boards. A single CIO board can be connected to up to 4 nodes within the hypercube. This allows different Mark IIIfp nodes to simultaneously place requests for accessing devices connected to a CIO board through its VME bus interface.

The operating system on the control processor is AT&T's Unix System V, Version 2.2. Routines have been provided to allow the interface between the control processor and programs running on the nodes of the Mark IIIfp under the control of the CrOS or Mercury operating systems described below. Detailed reference information suitable for Mark IIIfp applications development utilizing these operating systems is given in [3].

The Crystalline Operating System (CrOS) provides the programmer with the fastest possible communications scheme and is well suited for applications where the domain of the problem can be correctly mapped onto the hypercube topology. The most important characteristics of CrOS are:

- Communication is done between adjacent nodes only. Data transfers are only supported among "nearest neighbor" processing elements which are physically connected to one another.
- Communication is channel based. Each node designates a particular *channel number* to each of the links to its neighboring nodes in each dimension. CrOS communication routines make use of this channel number as their main parameter.
- Communication is synchronous. A request for exchange of data by one node must be matched by a corresponding request in the neighboring node, otherwise, a system deadlock would occur.
- High transfer rate. The bandwidth of this style of communication is 2 Megabytes per channel per second, with each node being capable of communicating data through all its communication channels simultaneously.

The Mercury operating system is used to allow asynchronous message passing between nodes. In addition, it is recognized that certain applications require asynchronous services, but which also contain certain portions which operate in a tightly coupled way and can benefit from the higher transfer rates available through the use of synchronous communication. In order to accommodate to these situations, the Centaur operating system was built on top of both CrOS and Mercury. Centaur gives the programmer extra flexibility by allowing the application program to switch between synchronous communication mode making use of CrOS services and asynchronous communication mode making use of Mercury services.

Support is provided for one user per Mark IIIfp hypercube, with a maximum of one main program running on each node of the system and a different program running on the control processor. Application programs running under CrOS may be written in C or FORTRAN. Those running under Mercury can currently be written in C only.

Applications: Many types of applications have been written or ported from sequential machines to run on the Mark IIIfp hypercube. Several are described in [2]. The following is a partial list:

- Electromagnetics Scattering Analysis:
 - Finite Difference
 - Methods of Moments
 - Finite Element

- Concurrent Database Implementations
- Image Processing
- Geophysical Modeling
- Lattice Gauge Simulations
- Computer Chess and Game Theory
- Vortex Flow Simulations
- Optimization Problems
- Lisp Interpreter
- Distributed Prolog
- Neural Network Simulations
- Neural Network Applications:
 - Early Vision
 - Pattern Recognition
 - Optimization
- Tracking Algorithms
- Battle Management Simulations
- Studies in Plasma Turbulence
- Plasma Particle Simulations
- Discrete Event Simulations
- Studies in Radiative Transfer
- Ray Tracing Algorithms

Electromagnetic Scattering by Finite Elements: In order to illustrate some of the issues involved in developing and running applications on a hypercube computer, we shall examine in detail an electromagnetic scattering code, which employs the technique of finite elements to obtain solutions. Early stages of this work, and several other parallel codes which solve scattering and related electromagnetic problems employing the method of moments, time-domain finite differences, and methods for frequency selective surfaces, are described in [4].

The well-posed scattering problem consists of a set of scatterers composed of dielectric and conducting materials with possibly anisotropic and inhomogeneous dielectric properties. A known electromagnetic wave illuminates the objects. The electromagnetic field scattered from the objects is to be calculated at infinity (the far field), and possibly close to the scatterers (the near field) as well. Since the finite element method requires a meshed problem space, some outer boundary condition is required which will impose an outgoing wave solution on the scattered field. Because of computational limitations, the outer boundary needs to be as close as possible to the scatterers. The wave equation for either the electric or magnetic field is solved in the frequency domain subject to the outer boundary condition, the boundary conditions imposed by the presence of conductors, and any simplifications allowed by the geometry.

The finite element mesh (nodes and elements) is generated to express the geometry of the scatterers, including their internal constituents. The node density and element types used to model the problems are chosen based on the specific accuracy required in the solution and the limits imposed by computational resources. The issue of mesh generation is a complicated one in itself. The density of nodal points and the complexity of element type will impact the accuracy of the solution and the computation required to compute that solution. The complexity of the outgoing wave boundary condition imposed at the outer problem boundary, as well as the position, shape, and grid density of the outer boundary also impact the solution accuracy.

We take as our basic equation the Helmholtz equation for either the electric or magnetic field in linear media, in the absence of free charges. The equation for \mathbf{E} is given by

$$\nabla \times (\mu^{-1} \cdot \nabla \times \mathbf{E}) - \frac{\omega^2}{c^2} \epsilon \cdot \mathbf{E} = 0 \quad (1)$$

where ϵ is a general electric permittivity tensor, μ is a general magnetic permeability tensor, ω is the angular frequency of the field, and c is the speed of light. The equation for \mathbf{H} is obtained by simply

swapping ϵ and μ throughout. In general, the dielectric functions may be complex, to model absorptive media. Either the \mathbf{E} or \mathbf{H} equation with boundary conditions is a sufficient statement of the problem.

The weak form equation required to employ the finite element method is obtained by taking the inner product of Eq. 1 with a test function \mathbf{T} and integrating over the problem domain. The test function must satisfy some basic constraints on continuity and integrability, but is otherwise arbitrary. Thus the weak or Galerkin form equation for the electric field is

$$\int_{\Omega} d^3x \left[(\nabla \times \mathbf{T}) \cdot \mu^{-1} \cdot (\nabla \times \mathbf{E}) - \frac{\omega^2}{c^2} \mathbf{T} \cdot \epsilon \cdot \mathbf{E} \right] = \int_{\Gamma} d^2x (\mathbf{n} \times \mathbf{T}) \cdot \mu^{-1} \cdot (\nabla \times \mathbf{E}) \quad (2)$$

where Γ is the contour which bounds the problem domain Ω . A similar equation may be obtained for \mathbf{H} .

In order to find the electric field solving Eq. 2, \mathbf{E} and \mathbf{T} must be restricted to a linear space of finite dimension. In the finite element method, this linear space is a finite summation over a set of basis functions \mathbf{w}_n ,

$$\mathbf{E} = \sum_n d_n \mathbf{w}_n \quad (3)$$

$$\mathbf{T} = \sum_n c_n \mathbf{w}_n \quad (4)$$

each of which is nonzero only in a small region of space. Interpolatory basis functions are preferred for most problems; these bases are defined to be unity at a single node of the mesh, and zero at all other nodes, typically based on simple polynomials defined in each polyhedral region of the mesh. The advantage of interpolatory basis functions is that each weight d_n in the summation for \mathbf{E} may be interpreted directly as the value of the solution at the corresponding node.

The assumption linking Eq. 1 and Eq. 2 is that the field \mathbf{E} which satisfies Eq. 2 for *any* test function \mathbf{T} (arbitrary within the space of functions satisfying certain criteria of continuity and boundary conditions) must be precisely that \mathbf{E} which solves Eq. 1. Substituting Eqs. 3 and 4 into Eq. 2 results in

$$\mathbf{c} \cdot \mathbf{Kd} = \mathbf{c} \cdot \mathbf{f}, \quad (5)$$

and the arbitrariness of \mathbf{T} implies that this must hold for arbitrary \mathbf{c} , therefore the \mathbf{d} must satisfy

$$\mathbf{Kd} = \mathbf{f}. \quad (6)$$

This system of equations is inverted to find \mathbf{d} , which determines the field everywhere in the computational domain according to Eq. 3. The matrix \mathbf{K} consists of integrals of the basis functions combined in pairs, with typical elements of the form

$$k_{ij} = \int_{\Omega} d^3x \left[(\nabla \times \mathbf{w}_i) \cdot \mu^{-1} \cdot (\nabla \times \mathbf{w}_j) - k_0 \mathbf{w}_i \cdot \epsilon \cdot \mathbf{w}_j \right] \quad (7)$$

Additional terms are required to implement certain boundary conditions, such as those at conductors, material interfaces and the computational domain truncated boundary. Eq. 7 (and the additional terms) imply several computationally important features of the system of equations to be solved: the matrix is symmetric, and extremely sparse, containing non-zero entries only when two basis functions share some portion of their non-zero domain. This sparsity, in combination with an appropriate parallel sparse equation solver on a hypercube computer, allows solution of problems with several hundred thousand unknowns.

Workstation Environment: In order to rapidly take advantage of the vast computational power of the Mark IIIfp supercomputer in an environment with a growing number of electromagnetic analysis parallel programs, we have developed an Electromagnetics Interactive Analysis Workstation, based on an Apollo DM4500 graphics computer connected to various Mark IIIfp hypercubes by a high-speed network. Some of the key features of the EIAW include a consistent and friendly interface based on a modular software design, transparent access to remote computing resources and an inherent ability for further expansion. The user is initially presented with a menu, enabling quick choices of graphical tools for model building and results display, interactive selection of parameters for a computation, a variety of electromagnetics codes, and an assortment of sequential or parallel computers which may be either local or remote. Fig. 2 shows the current location of the EIAW within our local area network. The network allows direct access to Sun, Counterpoint and Apollo machines as well as indirect access to several Mark IIIfp Hypercube systems. Program development is mostly done on Sun workstations, with executables being sent over the network to Counterpoint systems which currently act as host machines for the existent Mark IIIfp Hypercubes.

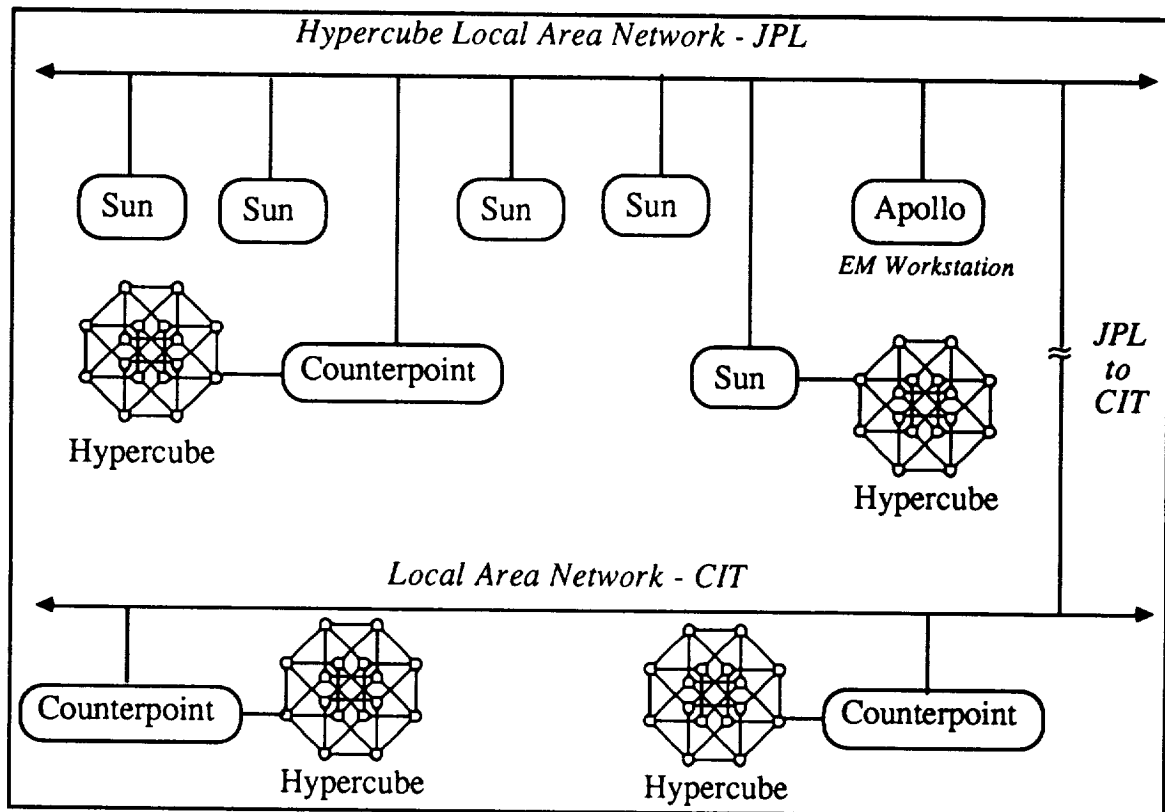


Figure 2. A sketch of the physical arrangement of the resources available through the JPL and CIT LANs.

Given a developed code, a design engineer may access the appropriate software and hardware resources for a given electromagnetic scattering problem from the workstation interface without needing to know anything about the network or the parallel computers. In a typical session, the engineer would start up the EIAW menu, and by simple mouse/cursor selection move through the design and testing of the scattering object as follows. First, the analysis tool, Finite Elements, is selected from a submenu from among such choices as FDTD (a Finite Difference Time Domain code) and Patch (a method of moments code). Next, a simple name for the object is chosen. The system will use this name, with various extensions, to create file names with a simple and consistent naming convention. Via another submenu, the engineer selects a model-building tool, which may be a commercially available CAD program. After the model is constructed, the model-building tool returns the engineer to the EIAW menu. Another submenu choice allows the engineer to enter model parameters such as the incident field description. The engineer then selects which computation engine to use to run the analysis program. At another menu command, the system partitions

the model into the appropriate number of pieces for the parallel computation, sends the model to the computation engine, runs the analysis program, and returns the solution field to the local environment. Post-processing utilities may then be selected to locally compute or display derived quantities of interest, such as the radar cross section of the object.

The Parallel Algorithms: The finite element method readily lends itself to parallel execution. The chief parts of the computational burden are the creation of the K matrix, and the solution of the sparse system of equations. The form of the integrand in Eq. 7 implies that non-zero entries in K arise from node couplings through polyhedral regions (elements) which contain a given pair of nodes. Each element couples all possible pairs of the nodes it contains, contributing entries in K which correspond to these node pairs. Because nodes are typically shared by several elements, a given node is coupled to all of the nodes in all of the elements of which it is a part, and only to those near-neighbor nodes. Therefore, a natural way to divide the problem is to divide the spatial domain. We give groups of neighboring elements and the nodes which belong to them to a single processor. Some nodes must be shared, which implies communication among the processors; thus it is advantageous to minimize the number of these shared nodes. In order to finish in the shortest time, each processor must do approximately an equal share of the work, and must therefore have an equal share of the model.

With this in mind, we have developed software to divide a model automatically and nearly optimally. We use an algorithm called Recursive Inertial Partitioning. The essential idea is to find the long axis of the model, and divide midway with a plane perpendicular to this axis. A moment computation is performed, corresponding to calculating the moment of inertia tensor, with each element having unit weight at its center. A plane perpendicular to the largest moment slices the object through the center of mass, with whole elements assigned to one side or the other. Each section is then redivided according the the same algorithm. Thus the object is divided into 2^D groups of elements, ready to be processed on a D -dimensional hypercube. Fig. 3 shows a sample finite element domain as divided by the RIP algorithm.

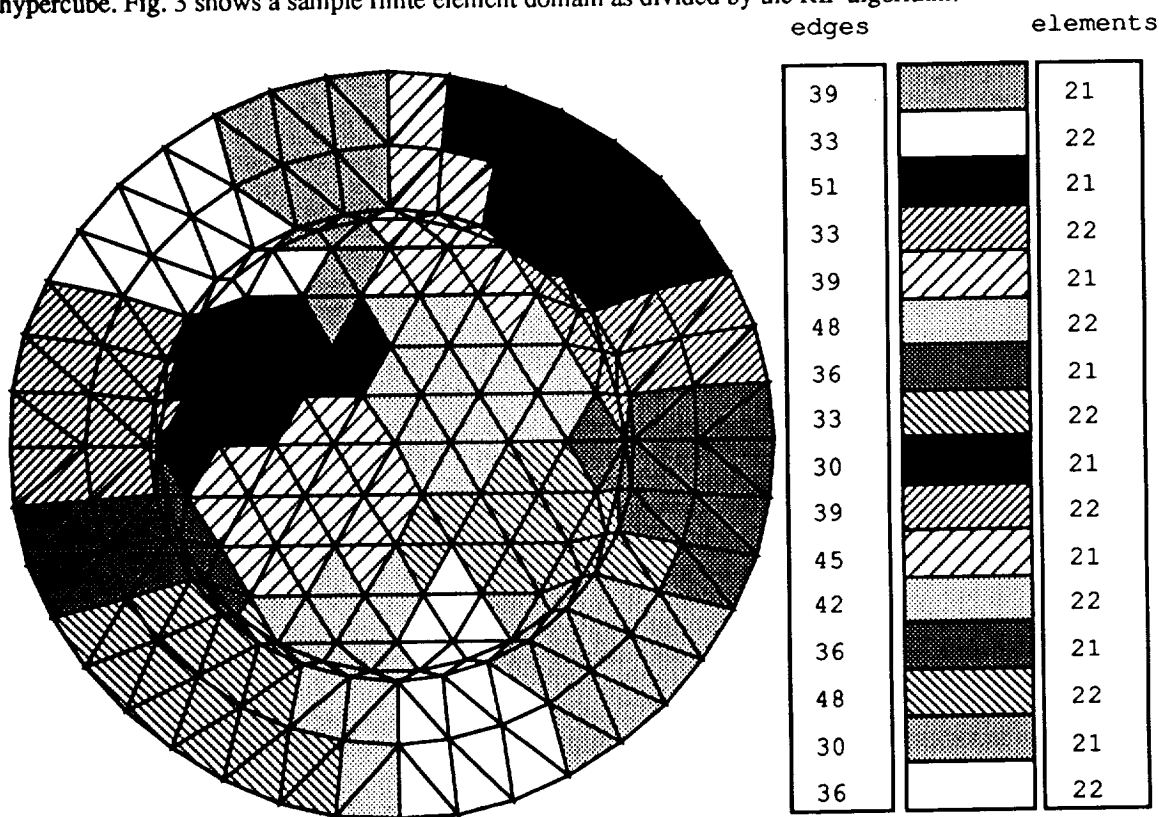


Figure 3 - Decomposition of a grid describing a cylindrical dielectric scatterer, using the RIP method implemented in the SLICE partition code. Load balance of elements and boundaries are illustrated.

Each processor of the hypercube computes its share of the K matrix completely independently, that is, with no interprocessor communication. This is achieved by redundantly storing the information for shared nodes in both processors, and by allowing the shared portions of the K matrix to remain unsummed. That is, when a K matrix entry involves two nodes each shared by two processors, that entry consists of a sum of terms, some of which are contributed by elements in the first processor, and some contributed by elements in the second. There is no need to explicitly sum these terms into one memory location, because the solution portion of the code can find all the appropriate partial sums when they are needed.

Solution is performed by a Preconditioned Bi-Conjugate Gradient algorithm. This is an iterative algorithm, with the drawback that the convergence to acceptable accuracy is not entirely predictable as to the number of steps. Convergence would occur with no error after N steps for a rank N system of equations, if the algorithm were performed with infinite precision arithmetic. In practice, well-posed problems typically converge to acceptable accuracy in less than $N/10$ steps. The great advantages of the method are that very little communication is required between the processors, and all steps of the algorithm can be done using the memory locations describing the system of equations. That is, in contrast to a banded-matrix solver, or (even better) a Crout decomposition, no matrix locations which are initially zero are filled in with non-zero values at any stage of the algorithm. Therefore, we store the non-zero matrix entries as an indexed list of numbers; we need never store zero entries.

The computationally significant parts of the PBCG algorithm are vector dot products and matrix-vector multiplies. Given distributed data storage as described above, two peculiar features of these operations as done in parallel are noted here. First, as to performing dot products, some vector components (corresponding to nodes in the finite element model which are shared among several processors) are replicated in several processors, while the contribution to the dot product from these components must be computed only once. Therefore, although these nodes are shared, only one processor has these nodes flagged as owned by that processor (the flagging having been done at the time of model partitioning). Partial dot products in each processor are computed using only vector components corresponding to owned nodes. Then the partial sums are globally added, a step requiring interprocessor communication. Second, with regard to matrix-vector multiplies, recall that K matrix entries corresponding to shared nodes consist of a sum of terms, and that these terms have not been explicitly summed, but rather reside in separate processors. The correct result may be obtained by having each processor do a matrix-vector multiply with respect to only the portion of the matrix and vector which reside in that processor, and then summing the results in an appropriate way with each processor contributing its portion (again, a step requiring interprocessor communication). In both of these operations, the amount of interprocessor communication is small compared to the amount of computation within each processor, provided the model consists of a large number of elements and nodes *per processor*.

Results: In order to demonstrate the type of object which may be modelled with the 2-D electromagnetic scattering code, and the sort of accuracy one may obtain easily, we compute the near and far field for a perfectly conducting circular cylinder with radius such that $ka = 50$ (i. e., the circumference is 50 wavelengths). The field is shown in Fig. 4. Since this is an object of simple geometry, one may also compute the solution analytically, and a comparison of the two solutions demonstrates the accuracy of the finite element solution. As seen in Fig. 5, good agreement (within 3 dB over a 30 dB range) with the analytic RCS was obtained by truncating the computational domain at $kr = 62$, while $kr = 56$ was not considered adequately accurate (errors exceed 4 dB). The more-accurate case involved less than 10,000 unknowns, which is a fairly small problem for the Mark IIIsp. With a 64 node Mark III, we can solve similar problems with over 200,000 unknowns, providing the ability to achieve either much higher accuracy, or to model larger and more complex objects.

The time to realistically solve a given problem depends on I/O rates as well as the time to set up and solve the finite element system of equations. Since I/O is currently done sequentially, there is a problematic bottleneck involved in reading the model and writing the fields from the hypercube. This I/O cost may be amortized by solving a large number of scattering problems based on the same model, such as by varying the incident field angle, the wavelength, or the properties of materials in the object. However, the I/O bottleneck can be significant when only one problem solution is needed, and we are investigating means of ameliorating the problem, such as generating the finite element mesh in parallel on the hypercube. Apart from I/O, the time required to solve a finite element problem is based on the time to set up the system of

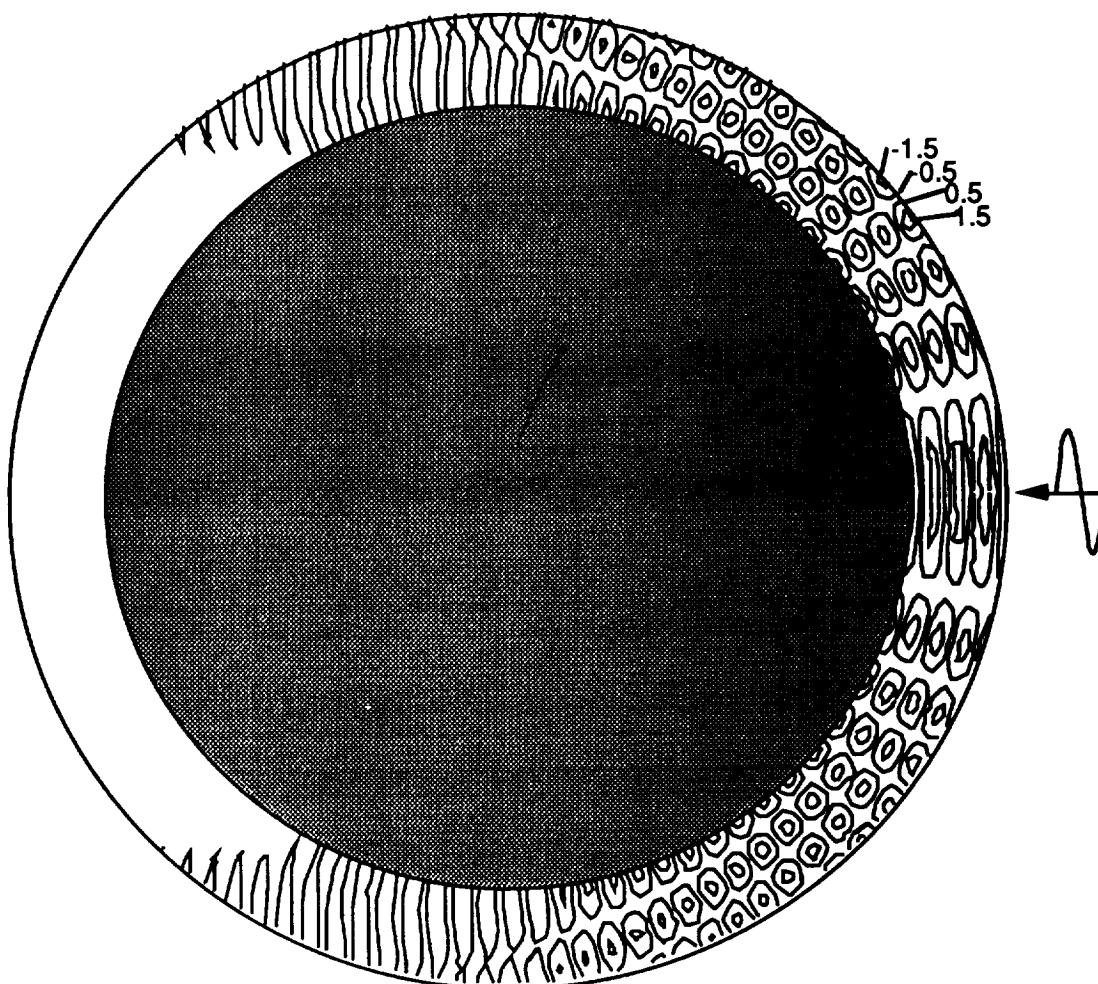


Figure 4 Contours of total field, real part for problem of scattering from $ka = 50$ conducting cylinder, TE case. The absorbing boundary is at $kr = 62$, about two wavelengths from the object. The incident field wavelength is shown to scale.

equations, and the time to solve the system. An experiment with a scaled model indicates the time required for finite element models on hypercubes of differing dimensions. We construct a series of models with the number of unknowns proportional to the number of processors in hypercubes of varying dimension: 1,2,4,8,16, and 32 processors, times about 4000 unknowns. Thus the largest model has about 128,000 unknowns in this experiment. We found the time for element set-up was 16 seconds in all cases: the set-up work scales linearly with the number of unknowns, and the parallel efficiency is nearly perfect, due to the lack of communication and the balance of the load achieved by the partitioning algorithm. The time to solve the system with the iterative solver varies somewhat erratically due to the indeterminacy of the number of iterations required; however, the time per iteration was also nearly constant in all cases, about 0.5 seconds (within 0.02 seconds tolerance). Due to communications overhead, there is a slight increase with increasing hypercube dimension, so that an iteration on the 32 node hypercube took about 6% longer than an iteration on a single node (which was processing only 1/32 as many unknowns).

We are developing a full 3-D scattering code. Extending a 2-D scattering code to handle full 3-D problems is not a trivial task. All three components of either the electric field or the magnetic field must be included in the model. Absorbing boundary conditions in 3-D are substantially different from those in 2-D, and are not widely published or tested. Proper physical conditions at boundaries of conductors, dielectric and magnetic materials are more complicated. Numerical stability of the method appears to be more difficult to guarantee

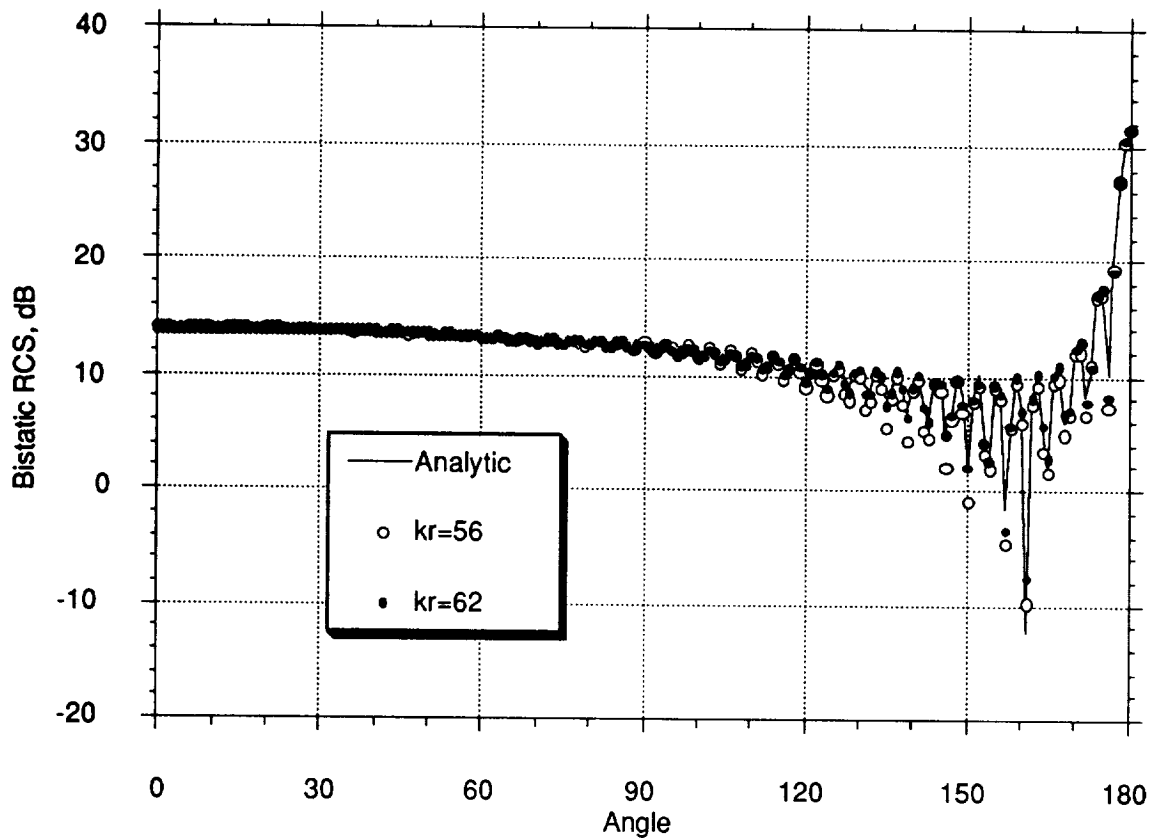


Figure 5 Bistatic RCS of TE scatter from $ka = 50$ conducting cylinder, modeled with 4 quadratic quadrilateral elements per wavelength. Solid curve is analytic solution; open circles, finite element result for absorbing boundary radius $kr = 56$; dots, finite element result for $kr = 62$.

than for the 2-D case, possibly requiring exotic vector basis functions and elements, such as so-called edge or tangential elements. Construction of consistent finite element meshes becomes substantially more difficult in 3-D, particularly for complicated objects containing varied materials. All of these difficulties are under investigation, and appear to be surmountable in the near future. The result will be a flexible environment for solving complicated 3-D electromagnetic scattering problems, relying heavily on the ability of parallel computers to solve such problems with hundreds of thousands of unknowns.

A Heterogeneous Application: The finite element code represents a large class of homogeneous problems, in which each node of the hypercube is doing the same type of task at the same time. The instructions in each node are not executed in lock-step, but essentially the same program is running in each processor, with separate data. Each processor works on the set-up at the same time, and then each processor works on its part of the system solution. We have also developed more heterogeneous parallel codes, such as a processor for synthetic aperture radar (SAR) images. In this arrangement, a 32 node hypercube is divided into 4 subcubes with 8 processors each. A separate code is loaded into each subcube, and the image sections are processed in pipeline fashion. Each subcube performs a portion of the processing functions, such as a range correlation fast Fourier transform, on part of the image, and then sends the image slice along to the next processing stage. The hypercube SAR system produces images at the rate of one 40 megabyte image every 43 seconds, utilizing the Weitek processors' ability to sustain 2.5 megaflops per second per processor.

Conclusion: The JPL/Caltech Mark IIIfp hypercube has proven to be an inexpensive, general purpose supercomputer. Despite the unavoidable difficulties of porting existing programs to a prototype computer,

including in this case having to deal with such issues as load balancing and communications, dozens of extremely diverse applications have been ported and are found to run efficiently on the hypercube. With expanding experience and the development of increasingly user-friendly software tools, the time required to develop hypercube applications will drop dramatically, with a corresponding explosion of applications available through public domain and commercial sources. As technology continues to provide faster microprocessors and the ability to do fast communications between processors, the potential speed of such applications on future machines patterned after the Mark IIIfp is nothing short of astonishing.

References:

1. Fox, G. C., Johnson, M. A., Lyzenga, G. A., Otto, S. W., Salmon, J. K., and Walker, D. W.: Solving Problems On Concurrent Processors. Volume I. General Techniques and Regular Problems. Prentice Hall (Englewood Cliffs, New Jersey), 1988.
2. Angus, I. G., Fox, G. C., Kim, J. S., and Walker, D. W.: Solving Problems On Concurrent Processors. Volume II. Software for Concurrent Processors. Prentice Hall (Englewood Cliffs, New Jersey), 1990.
3. JPL internal document: Hypercube Project Programmer's Manual. JPL D-3220, Rev. D, 1988.
4. Calalo, R. H., Imbriale, W. A., Jacobi, N., Liewer, P. C., Lockhart, T. G., Lyzenga, G. A., Lyons, J. R., Manshadi, F., and Patterson, J. E.: Hypercube Matrix Computation Task. Report for 1986-1988. JPL Publication 88-31, 1988.